Accelerating Asynchronous Programs through Event Sneak Peek

Gaurav Chadha   Scott Mahlke   Satish Narayanasamy
University of Michigan, Ann Arbor
{gauravc,mahlke,nsatish}@umich.edu

Abstract

Asynchronous or event-driven programming is now being used to develop a wide range of systems, including mobile and Web 2.0 applications, Internet-of-Things, and even distributed servers. We observe that these programs perform poorly on conventional processor architectures that are heavily optimized for the characteristics of synchronous programs. Execution characteristics of asynchronous programs significantly differ from synchronous programs as they interleave short events from varied tasks in a fine-grained manner.

This paper proposes the Event Sneak Peek (ESP) architecture to mitigate microarchitectural bottlenecks in asynchronous programs. ESP exploits the fact that events are posted to an event queue before they get executed. By exposing this event queue to the processor, ESP gains knowledge of the future events. Instead of stalling on long latency cache misses, ESP jumps ahead to pre-execute future events and gathers useful information that later help initiate accurate instruction and data prefetches and correct branch mispredictions. We demonstrate that ESP improves the performance of popular asynchronous Web 2.0 applications including Amazon, Google maps, and Facebook, by an average of 16%.

1. Introduction

Asynchronous or event-driven programming [3] has become the dominant programming model in the last few years. In this model, computations are posted as events to an event queue from where they get processed asynchronously by the application. A huge fraction of computing systems built today use asynchronous programming. All the Web 2.0 JavaScript applications (e.g., Gmail, Facebook) use asynchronous programming. There are now more than two million mobile applications available between the Apple App Store and Google Play, which are all written using asynchronous programming. Distributed servers (e.g., Twitter, LinkedIn, PayPal) built using actor-based languages (e.g., Scala) and platforms such as `node.js` rely on asynchronous events for scalable communication. Internet-of-Things (IoT), embedded systems, sensor networks, desktop GUI applications, etc., all rely on the asynchronous programming model.

We consider the Web 2.0 applications as a case in study. Web applications are increasingly popular today as it allows users to easily run them within their web browsers and access data stored in the cloud. These web applications, written in a scripting language (mostly JavaScript), receive asynchronous input from diverse input sources such as user clicks, servers over the network, microphone, camera and other sensors. Web development platforms use the asynchronous or event-driven programming model as it is a natural fit for handling this rich array of asynchronous input.

In spite of the ubiquitous use of asynchronous programming, today's general-purpose processor architectures are heavily optimized for the synchronous programming model, and ignores the unique execution characteristics of asynchronous programs. As the processor industry continues to embrace heterogeneous processor designs with domain-specific accelerators as an answer to the end of Dennard scaling, we envision that at least a few of the processor cores could be customized for efficiently executing the asynchronous programs. To this end, this paper proposes a novel processor optimization for accelerating asynchronous programs.

Asynchronous programs present unique challenges and opportunities for performance optimization. Unlike a synchronous program that executes one long (billions of instructions) task after another, an asynchronous program's execution consists of many short (only millions of instructions) events orchestrated by the timing of external events (e.g., user click). Such fine grained interleaving of many small and varied tasks destroys instruction and data locality, and exposes little repeatable patterns assumed by processor components such as the branch predictor. Cold misses, which are of little concern in synchronous programs, constitute a significant source of pipeline stalls in asynchronous programs with short events. Finally, asynchronous programs, especially web applications, tend to exercise large instruction footprints to support a rich set of features, resulting in significantly higher instruction cache miss rates than synchronous programs.

Fortunately, asynchronous programs also provide us with new opportunities for optimizations that are absent in synchronous programs. In an asynchronous program, events are

ISCA '15, June 13 - 17, 2015, Portland, OR, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3402-0/15/06$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750373

enqueued in an event queue before they are processed sequentially by a thread. By exposing the event queue to the processor, the processor can know the sequence of events that are going to be executed in the future. Our insight is that this knowledge can be exploited to take a "Sneak Peek" into the executions of the future events and gather information about them ahead of time. This information can then be used to significantly improve the accuracy of various hardware predictors when an event is executed.

We propose the Event Sneak Peek (ESP) architecture that exploits the above insight to significantly reduce instruction and data cache misses, and branch mispredictions for asynchronous programs. On encountering a long latency last-level cache (LLC) miss for instruction or data, ESP uses the idle CPU cycles to "jump ahead" to the next event in the event queue and speculatively pre-execute it. While pre-executing an event, ESP records the sequence of instruction and data cache blocks accessed, and also the branch mispredictions. ESP jumps back to the normal execution once the LLC miss gets resolved. Later, during the normal execution of a pre-executed event, the previously recorded information is used to initiate accurate prefetches and correct branch mispredictions.

Pre-execution of future events is speculative as they may depend on the earlier skipped events. However, since the events perform varied tasks, they are fairly independent of each other. Thus, speculative pre-execution closely matches the normal execution and helps collect accurate predictions. ESP does not use the computed results from the speculative pre-execution of event, as it may require fairly complex hardware such as the components used in Thread-Level Speculation (TLS) [19, 23, 30, 31] to check for dependencies between events and recover from mis-speculations. ESP can exploit the above characteristics of events present in a large number of asynchronous applications - mobile and web 2.0 applications, desktop GUI applications, IoT, sensor networks. However, events in web servers and `node.js` applications are either independent or do not hold up processing of future events, making it easier to expose event-level parallelism, without needing ESP.

We devise several architectural solutions to realize an efficient ESP design. First, ESP can jump ahead multiple events. This is useful when an LLC miss is encountered while pre-executing an event after having already jumped earlier. Thus, as long as there are events in the queue, the processor can utilize the idle cycles due to LLC misses for pre-executing future events.

Second, an event may encounter several LLC misses, offering several opportunities to jump ahead to the next event. To avoid always jumping to the beginning of the next event, ESP provides the capability to save the execution context of a pre-executed event and return to that intermediate state on the next LLC miss.

Third, ESP employs small instruction and data cachelets (L0) for exclusive use during speculative pre-execution of events. Cache blocks accessed during pre-execution are brought directly to the cachelets, skipping L1 and L2, and they are never written back. These cachelets help ensure correctness by isolating the speculative store updates during a pre-execution from the normal event execution. They also help improve performance by not polluting the L1 and L2 caches for the normal event execution. Furthermore, they support efficient event pre-execution by retaining its working set, even when the processor control switches back and forth between normal and speculative pre-executions.

Finally, as ESP skips potentially thousands to millions of instructions when it jumps to the next event, fetching cache blocks directly into L1/L2 during an event's pre-execution would be too early, and hence wasteful. To address this problem, ESP uses hardware lists for collecting cache block addresses and branch mispredictions in a compressed format during an event pre-execution. The recorded information is later used during event's normal execution to initiate timely prefetches and correct branch mispredictions.

ESP could be considered as a form of runahead execution [16, 7, 26, 25], though there are some fundamental differences. Runahead execution pre-executes independent instructions that follow only a LLC *data* cache miss. Since runahead would stall on an instruction cache (I-cache) miss, it does not help improve the I-cache performance. In contrast, on encountering an instruction cache miss, ESP can skip the current event, and start fetching instructions from a different location for the next event. Second, runahead is limited by the number of independent instructions that it can discover after a load miss. ESP is more effective in finding independent instructions to pre-execute, as events tend to be independent of each other. Finally, while runahead pre-executes instructions that immediately follow a load miss, ESP may skip millions of instructions as it jumps ahead to the next event. This mandates several new solutions discussed earlier.

We demonstrate the utility of the ESP architecture for accelerating the Web 2.0 applications, but it should be more broadly applicable for optimizing other event-driven systems. Our benchmark suite consists of a popular representative website for each type of web applications: e-commerce (`amazon`), interactive maps (`google maps`), search (`bing`), social networking (`facebook`), news (`cnn`), utilities (`google docs`), and data-intensive applications (`pixlr`).

We show that ESP can improve the performance of a baseline architecture with a next-line instruction prefetcher and our implementation of Intel's data prefetchers (next-line and stride) [15] by 16%, whereas runahead improves it by only 6.4%. ESP achieves this by reducing L1 I-cache misses per kilo-instructions (MPKI) from 17.5 to 11.6, L1 data-cache miss rate from 3.2% to 1.8%, and branch misprediction rate from 9.9% to 6.1%. Since ESP executes additional instructions during pre-execution of events, it increases energy overhead by 8%.

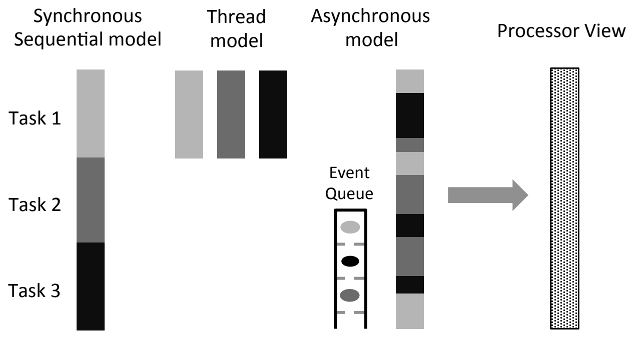


Figure 1: Comparison of Programming models. A processor's simplistic view of an asynchronous program's execution.

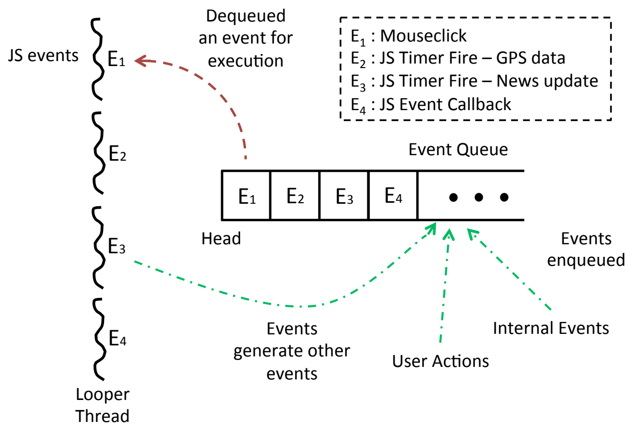


Figure 2: Asynchronous execution of a web application.

2. Background and Motivation

This section distinguishes asynchronous programs from synchronous programs, and discusses why they perform poorly on conventional processor architectures. It also describes asynchronous web applications, which we use in our experiments to demonstrate the effectiveness of the ESP architecture.

2.1. Asynchronous Vs Synchronous Programming

For the purpose of illustration, let us consider a program consisting of three distinct tasks shown in Figure 1. A typical single-threaded synchronous program completely finishes a task before starting another. In a multi-threaded synchronous program, the tasks can be assigned to different threads. These threads may either run concurrently on different processor cores as shown in Figure 1, or the OS may interleave some of them at a coarse granularity on a single core. In both models, as the same task is executed for long periods of time (typically billions of instructions), they exhibit a high degree of locality and repetitive behavior, which current processors exploit to achieve high cache performance and branch prediction accuracy.

In computing systems with frequent long latency blocking operations (mostly due to I/O), synchronous programs waste processor resources waiting for those operations to complete. An asynchronous program avoids this problem by splitting a task into multiple sub-tasks at the boundaries of blocking operations and associates them with event conditions. When the required event condition is met (e.g., user click), the corresponding sub-task (event handler) is enqueued into an event queue. Instead of blocking on an I/O operation, an asynchronous program switches to execute a ready sub-task from the event queue. Programmers also take care to ensure that each event execution is short, because otherwise the system may become unresponsive.

Such a fine-grained interleaving of short (only millions of instructions) sub-tasks (events) from different tasks (Figure 1) destroys instruction and data cache locality in asynchronous programs. Also, it exposes little repetitive behavior, which is critical for achieving prediction accuracy. Unfortunately, conventional processors are presented with a simplistic homogeneous view of an asynchronous thread (Figure 1), which is far removed from reality. In this paper, we propose to remedy this inconsistent view of the processor, by exposing the event queue and the event execution boundaries, and use that knowledge to improve locality and branch prediction.

2.2. Illustration: Asynchronous Web App's Execution

Figure 2 shows an example execution of an asynchronous web application. The looper thread in the renderer process of a web browser, constantly polls an event-queue. It dequeues one event at a time and invokes the necessary JavaScript (JS) handler function to process the event. The events may be generated internally or in response to an external input. If an event has to wait for any long-latency operation (e.g., downloading a web page), instead of stalling, the event handler would register an event callback to be invoked when the long-latency operation has completed, and then return.

The events get queued as soon as they are generated. However, we observe that they remain in the event-queue for several tens of microseconds before getting dequeued and executed. This duration can be used to pre-execute the future events, which can help improve their performance when they finally execute. Also, as shown in Figure 2, events perform varied tasks, and as a result exhibit high degree of independence. This property allows us to speculatively pre-execute future events with high accuracy.

2.3. Asynchronous Web 2.0 Applications

Web 2.0 has revolutionized the Internet experience. This has been enabled in large part by the shift of computation from servers to clients. This shift, along with the increased complexity of web sites in an effort to provide a rich user experience has dramatically increased the computation responsibilities of the clients (mobile and desktop). As a result, the proportion of JS execution in the renderer process has almost doubled in the last ten years during the transition from Web 1.0 to 2.0, and is only likely to grow further.

Asynchronous execution of client-side JS is responsible for, among other things, handling user interaction with the web

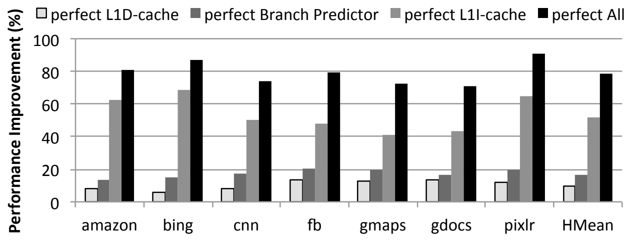


Figure 3: Performance potential in web applications.

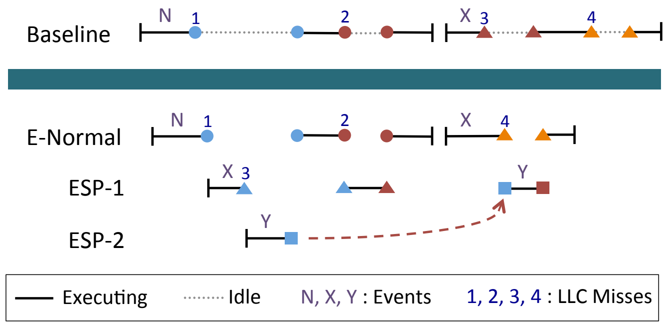


Figure 4: Illustration: ESP Execution

browser and providing dynamism to the web page content. It commands a lion's share of the client-side computation, and directly affects browser response time as it can block further user interaction. Even in web applications that require network transactions, JS computation accounts for more than 20% of response time [13]. Thus, improving JS performance in web browsers is not only useful for compute-intensive web applications, but can also help improve response time.

For the reasons stated in Section 2.1, asynchronous web application incur very high L1-I cache miss rates and branch misprediction rates, with moderate L1-D cache miss rates. By using perfect caches and branch predictors, we can nearly double the performance of asynchronous web applications (Figure 3).

However, existing prefetching techniques are either incapable of alleviating the problem or come at too big a hardware cost. This is because, the large instruction footprints of event-driven applications forces instruction prefetchers to keep track of a large amount of state, exploding their hardware overheads. Also, as there are less repeatable instruction access patterns to be exploited in asynchronous programs due to small events executing disparate code, conventional instruction prefetchers perform poorly. Our goal is to expose the event-queue to the hardware and use that information to accelerate asynchronous applications.

3. Event Sneak Peek (ESP): Design

Conventional microarchitecture designs have ignored the unique challenges and opportunities posed by asynchronous programs. This section presents our insights for exploiting event-level parallelism in asynchronous programs for improving ILP. We then present an overview of the important components in our Event Sneak Peek (ESP) architecture, which exploits our insights to accelerate asynchronous programs.

3.1. Exploiting Event-Level Parallelism

By exposing the software event queue to the hardware, the processor can know the events that are going to be executed in the future. Since events are fairly independent of each other, on encountering a long latency LLC (last-level cache) miss while executing an event, instead of stalling the pipeline, the processor can "jump ahead" to speculatively pre-execute the first event waiting in the queue. If another LLC miss is triggered while speculatively pre-executing an event, the processor can jump ahead one more time to the second event in the queue. As we later show in Section 6.6, we did not find many opportunities to pre-execute a third pending event in the queue. Therefore, we made an important decision to support only two event jump aheads at any time. Eventually, when the LLC miss is resolved, the processor can jump back to continue the normal execution.

Pre-execution of a future event is speculative, because there is a small chance that it might be dependent on the previous events that were jumped over. Therefore, if we want to reuse the computation from the pre-execution of an event, we would have to invest in some fairly complex hardware to check for any dependencies between events and recover from mis-speculations.

Instead, we design a simpler optimization where speculative pre-execution of an event is used to only collect the instruction and data addresses it accessed, and the branch outcomes. The collected information is later used when the event is executed in the non-speculative mode to initiate accurate prefetches and correct branch predictions. In this way, we exploit event-level parallelism in asynchronous programs to improve their ILP.

3.2. Event Sneak Peek: Illustration

Figure 4 illustrates an example execution in ESP. It shows a state where the processor is executing the current non-speculative event `N`, in the normal mode (`E-Normal`). On encountering an LLC miss (`1`) for `N`, ESP skips the current event and jumps ahead to speculatively pre-execute the first waiting event (`X`) in the queue. We refer to this execution mode as `ESP-1`. If another LLC miss (`3`) is encountered while pre-executing the event `X`, or if the event `X` ends, the processor jumps ahead one more time to the `ESP-2` mode, where it pre-executes the second event in the queue (`Y`). The instruction and data addresses accessed, along with the branch outcomes, during the ESP modes are recorded. Eventually, when the LLC miss resolves for the normal event `N`, the processor goes back to continue its non-speculative execution.

If the normal event `N` encounters another LLC miss (`2`), the processor again jumps ahead to `X` and starts pre-execution from the point where it had left off during the last visit.

When the current event `N` ends, the software dequeues `X` and enqueues `Z`. Thus, the event `X` is now the current non-speculative event, and the events `Y` and `Z` correspond to the

ESP-1 and ESP-2 modes respectively. The prediction information gathered for X during its pre-execution is used to improve prefetching and branch prediction in the normal mode. Thus, event X finishes earlier in ESP, compared to the baseline processor.

3.3. Design Overview

Figure 5 illustrates the Event Sneak Peek (ESP) architecture. ESP consists of a hardware event queue that holds the event handlers (starting instruction addresses) for the future events. Software is responsible for enqueing and dequeing event handlers into this queue through two new instructions that we add to the ISA.

The rest of the ESP's architectural extensions can be broadly classified into three main categories. The first set of components help preserve the execution contexts of the current event and the speculatively pre-executed future events. The second set of components help collect branch outcomes and addresses accessed during the pre-execution of an event in the sneak peek mode. The final set of modifications enhance existing prefetchers and branch predictor with the information collected by pre-executing events. The rest of this section provides an overview for each of these three categories.

3.4. Persisting Event Execution Contexts

ESP supports speculative pre-execution of future events to be re-entrant. It is possible for the current event to encounter several LLC misses. When the execution jumps ahead to the next event on an LLC miss, instead of starting from the beginning of the event, the pre-execution continues from the point where it was suspended during the last visit. This allows ESP to pre-execute deep into future events.

Similar to coarse-grained multithreading (CGMT), ESP maintains additional execution contexts (registers, caches, and global branch history), to allow control to switch back and forth between the non-speculative current event and the speculative future events. The number of additional execution contexts required depends on the number of events that ESP allows to jump ahead at any given time, which is limited to two as we mentioned earlier (Section 3.1).

Registers: The baseline register context supports the current non-speculative event. In addition, ESP supports two sets of additional register contexts, where each set includes a retirement register alias table (RRAT), and special purpose registers (e.g. program counter (PC), stack pointer (SP)).

I and D-Cachelets: A naïve ESP design would utilize the caches in the baseline processor to support event pre-executions. However, this poses both **correctness** and **performance** challenges. To ensure correctness, a store pre-executed by a future event in an ESP mode should not be allowed to update the cache and memory state, because otherwise the earlier non-speculative event may see an incorrect value. Simply dropping the speculative store updates during pre-execution is also not a feasible solution, because then the loads during pre-execution may not return correct values, degrading the accuracy of information collected about events in ESP modes.

To solve this problem, ESP has two small data cachelets (D-cachelets), one for each ESP mode, which act as the L0 cache used exclusively in the ESP modes. Stores in the ESP modes update the corresponding D-cachelets. However, cache blocks in D-cachelets are not written back to L1 and beyond, which ensures correct memory state for the earlier non-speculative event.

D-cachelets provide significant performance benefits. A cache block fetched to service a load or a store in an ESP mode, bypasses the caches and is brought directly into the corresponding D-cachelet. This avoids polluting the L1 cache state for the non-speculative event and the D-cachelet state for the other ESP mode. D-cachelets also help improve performance in the ESP modes by preserving the working set for the pre-executed events as the control switches back and forth between the normal and ESP modes.

Since instruction caches are read-only, we do not need instruction cachelets (I-cachelets) for correctness. However, ESP uses two I-cachelets for the two ESP modes to reap the performance benefits noted above for D-cachelets.

Branch Predictor Context: Our baseline processor models Pentium M branch predictor [35], which uses a Path Information Register (PIR). We find that preserving the small PIR states across control switches between events can result in significantly more accurate branch predictions. Therefore, ESP uses two additional PIRs for the two ESP modes. Preserving the rest of the branch predictor state did not improve prediction accuracy significantly enough to warrant the high area cost it would entail. Thus, the rest of the branch predictor structures are updated just like in the baseline processor in both the normal and the ESP modes.

3.5. Prediction Lists

To reduce cache miss rate, simply holding the accessed cache blocks in the cachelets during the ESP modes is not a viable option, since the cachelets (I and D) are designed to be large enough to only hold most of the current working set of the pre-executed events, so that the events in ESP modes can execute faster. The instruction and data working sets of events can be 100s of KBs, and are too large for the cachelets or for the L1 caches.

Cache blocks with read-only permissions could be brought to the L2 cache. However, bringing these cache blocks even to the L2 is premature, especially in the case of a long running current event. Moreover, this only solves part of the problem, since when these cache blocks are accessed by the current event, the processor still has to pay the penalty of an L2 cache access.

Therefore, ESP uses two sets of two hardware lists (**I-List** and **D-List**) to record the cache block addresses accessed during the two ESP modes. The records also contain the time (measured as instruction count from the beginning of the

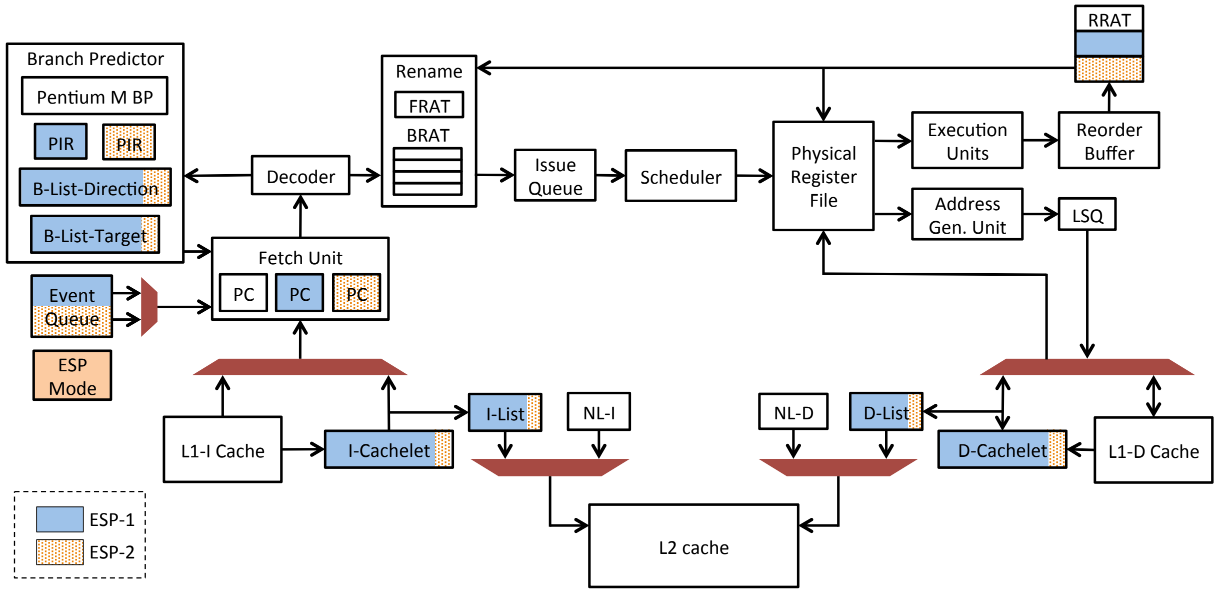


Figure 5: The ESP Architecture

event) of those cache accesses. Timely prefetches to these cache blocks are then made, when the event executes in the normal mode.

To increase the performance of the branch predictor, simply training it on branches executed during the ESP modes does not work well. This is because, this way the branch predictor gets trained on branches from three different events, intermittently, polluting its state. Also, this training happens so prematurely that other branches would have overridden it by the time it is needed. However, if we are able to keep the branch predictor trained on branch outcomes of just enough future branches, it will have trained on the most relevant branches to perform well.

To this end, the instruction addresses and directions of branches are stored in a list (**B-List-Direction**), along with the targets of taken branches (**B-List-Target**). Similar to the I and D-lists, B-List-Direction records the time of the branches to more accurately guide branch prediction during the normal mode.

An alternative design would be to use the branch directions and targets recorded in B-lists directly, matching the prediction with the specific branch, bypassing the branch predictor completely. However, this requires lining up the entries in the B-lists with the corresponding dynamic instances of branches. This is a non-trivial task, since a difference of a single branch between the speculative pre-execution and the normal execution would mis-align the predictions and the branches, rendering the B-lists useless.

Though, previous work [38] successfully used software analysis and extra information stored in hardware with each branch, to correct misalignment of predictions with branches, a similar technique would be infeasible in our case. This is because, the length of the speculative slice in [38] is much shorter compared to ESP. This is critical, since the efficacy of their technique degrades with increasing number of control flow paths. Also, the hardware overhead increases with the length of the speculative execution. Moreover, unlike ESP, any misalignment would impact only that short speculative slice. Owing to these difficulties, this approach was not used in ESP.

The lists for the ESP-1 mode are much larger than the lists for the ESP-2 mode, as its pre-execution can go much deeper.

3.6. ESP Predictors

When an event executes in the normal mode, information that was previously recorded in an ESP mode is used to prefetch and correct branch mispredictions.

To ensure timely prefetches, ESP issues prefetches from the I-list and D-list a preset number (190) of instructions in advance of its use (determined by the instruction count stored in the list entry), or the earliest possible. One challenge is initiating prefetching before an event starts so that cache misses during the initial execution of an event are avoided. We solve this problem by exploiting our observation that a looper thread (one that dequeues and processes events from the event queue) executes extraneous instructions (about 70) for event queue management towards the end and also before beginning an event. We use this opportunity to start initiating prefetches at least 70 instructions before an event starts.

These prefetch requests override those from the baseline prefetcher, next-line (NL). Once the addresses in the list are exhausted, NL takes over. Therefore, for long events, ESP would initially use the lists issuing accurate prefetch requests, but later has to rely on the baseline prefetcher.

The instruction addresses and branch outcomes stored in B-List-Direction, along with the targets in B-List-Targets, are used to train the branch predictor with several branches ahead of their execution. The training is kept loosely coupled with the actual branch execution, a preset number of branches ahead, so that the branch history is neither too far in the future nor too short, so as to enable accurate branch prediction. Once the list entries are exhausted, the branch predictor works as it normally would, without the help of just in time training.

4. Implementation Details

This section first describes how the different architectural extensions of ESP interact with each other to realize this design, followed by the details of the added hardware components. It concludes with a description of rare cases that arise during ESP modes and how they are handled.

4.1. Switching Event Execution Contexts

The events to be executed subsequently are present in a software event-queue. We propose that arguments to the event handler be passed as data members of an object, whose address is passed to the event handler. With this change, the software event-queue is exposed to the hardware in a 2-entry register-like structure which keeps track of the next two events. Each entry of this *Event Queue* holds the starting address of an event handler, the argument object address and an execution-underway (EU) status bit, which indicates if speculative pre-execution of this event is already underway or not. We propose addition of function intrinsics to the system to enqueue and dequeue events from the Event Queue.

The `ESP-mode` status register specifies the event that is being executed, which can either be the non-speculative current event (normal mode) or one of the future events in the hardware event queue (ESP modes).

Entering ESP mode: The processor enters ESP-i mode when a memory operation misses in the last-level cache (LLC) and the corresponding instruction reaches the head of the Re-Order Buffer (ROB), where "i" is 1 on switching from the normal mode, and 2 on switching from the ESP-1 mode (these values of "i" have been used in the rest of the section to describe the hardware components used).

The address of the instruction that caused the LLC miss is recorded in the duplicated Program Counter register for ESP-i, PC_{ESP-i}. This instruction and all following instructions of this mode are drained from the pipeline without committing them, similar to how wrong-path instructions in the case of a branch misprediction are handled.

To correctly resume execution on return to a mode, ESP preserves the register state of each context, by duplicating the Retirement Register Alias Table for each context, $RRAT_{ESP-i}$, while using one common physical register file. While the pipeline is being drained, before entering ESP-i mode, RRAT_{ESP-i} is copied onto the Front-end RAT (FRAT), which restores the register state of ESP-i mode.

Execution in ESP mode: If the event is starting its execution in the ESP-i mode (EU bit is not set), the address of the instruction to fetch is specified by the corresponding Event Queue (and the EU bit is set). On the other hand, if it is resuming execution, PC_{ESP-i} holds the next instruction address.

In ESP-i mode, instruction fetches use the *I-cachelet*$_{ESP-i}$, as an L0-I cache, bypassing the L1-I and L2 caches. Addresses of I-cache blocks fetched by ESP-i are stored, in order, in *I-list*$_{ESP-i}$. These are later used for instruction prefetching when the event executes in the normal mode. Data requests use their counterpart structures in exactly the same fashion.

Exiting ESP mode: When the LLC miss of the current event is resolved, the processor returns to its current execution. Similar to the manner in which a branch misprediction is handled, all instructions in the pipeline are flushed at this point, in an effort to return to the execution of the non-speculative event as quickly as possible. To restore the register context, RRAT of the current event is copied on to the FRAT. The Return Address Stack (RAS) is cleared, since it might contain return addresses of functions in ESP-i modes.

4.2. Enhancing Cache Performance

I and D-cachelets isolate cache blocks accessed during the ESP modes from the rest of the system for correctness and performance purposes, as explained earlier. On the other hand, I and D-lists record cache block addresses and enable timely prefetches enhancing cache performance.

Cachelets: The sizes of the two cachelets are provisioned to be large enough to capture 95% of reuse in the ESP modes. Both I and D-cachelets are 12-way set associative caches, of size 6 KB each. Since the processor spends much less time in the ESP-2 mode (two event jump ahead) compared to the ESP-1 mode (one event jump ahead), its cachelet can be a lot smaller than the one used for ESP-1 mode. Through empirical analysis (Section 6.6), we find that 5.5 KB I-cachelet for ESP-1 mode, and 0.5 KB I-cachelet for ESP-2 mode is adequate (same for D-cachelets).

To isolate the two speculative events from each other, one way of the cachelets is reserved for ESP-2 (first way). When the current event finishes execution, the event in ESP-2 mode moves to ESP-1 mode, while the next event in the Event Queue will be executed in ESP-2 mode. The last way is now reserved for ESP-2. The event now in ESP-1 mode, gets ten more ways of cachelet space, plus its original reserved way. In this manner, the way reserved for ESP-2, switches between the first and the last way, on completion of events.

Lists: The I and D-lists are used to store the corresponding cache block addresses accessed during ESP modes, such that those cache blocks can then be prefetched when the event executes in the normal mode.

I-list stores the list of I-cache block addresses accessed by retiring instructions in the ESP modes. Each entry is composed of an I-cache block address stored as an offset from the previous list entry (8 bits), the number of contiguous cache blocks accessed after it (3 bits), the number of instructions executed before accessing this block, stored as an offset from the previous list entry (7 bits) and a large offset bit indicating that this cache block is much further away from the previous list entry than can be encoded in an offset of 8 bits. In such a case the next two list entries specify the complete 26-bit cache block address.

The list is physically divided in to two circular queues, 499 bytes for ESP-1 and 68 bytes for ESP-2. When the current event finishes execution, the event executing in ESP-2 mode, moves to ESP-1. It now uses I-list$_{ESP-1}$. However, it already has I-cache block addresses in I-list$_{ESP-2}$. These are copied onto I-list$_{ESP-1}$, before its head, such that the last element in I-list$_{ESP-2}$ is before the head of I-list$_{ESP-1}$. The event in normal mode of execution reads the contents of I-list$_{ESP-1}$ starting from its head, while the event in ESP-1, stores I-cache block addresses starting from an entry right after the last element copied from I-list$_{ESP-2}$.

D-list has an identical composition as I-list, except that the two circular queues are, 510 bytes for ESP-1 and 57 bytes for ESP-2.

4.3. Augmenting the Branch Predictor

The branch predictor modeled in our design, uses PIR to index into multiple different tables (e.g. global predictor table). We explored different design choices, as explained in Section 6.5, and finally settled with a separate PIR for each execution context, while sharing the rest of the branch predictor structures.

It is, however, augmented with **B-List-Direction** and **B-List-Target**. Each entry in B-List-Direction stores the instruction address of a branch as an offset from the previous list entry (4 bits), direction of the branch (1 bit) and whether it is an indirect branch (1 bit). The first two entries out of every thirty, store the retired instruction count as an offset from the last instruction count.

B-List-Target stores the branch targets of taken indirect branches. Each entry is composed of the branch target stored as an offset from the instruction address in B-List-Direction (16 bits) and a bit indicating if the address could be expressed using the offset. If not, the next two entries specify the complete branch target address.

4.4. Miscellaneous

The cachelets do not participate in coherence, keeping any updated data values private to the speculative execution context.

Any exceptions raised during ESP modes, are silently dropped, and the execution continues ignoring them. This is fine, since the execution is speculative.

Data updates in ESP modes are only stored in the D-cachelets. If a dirty cache block gets evicted from the D-cachelets, those updated data values are lost. The speculative pre-execution, thereafter, uses the older data values present in either the L1-D cache or lower levels of the memory hierarchy. This can, potentially, lead to an incorrect path of execution, generating incorrect hints. This lowers the performance improvement possible, but does not cause correctness problems.

4.5. ESP for any Asynchronous Program

We have described ESP in the context of web 2.0 applications running in a web browser, which had one looper thread dequeueing and executing events from a single event queue. In the general case, there could be multiple looper threads executing events from multiple event queues. In these systems, the software runtime arbitrates between the different event queues and looper threads, and decides what event runs on which looper thread.

In such a case, each looper thread would run on a separate ESP core. When determining the event to run on a looper thread, the software runtime, will now, additionally, predict the next two events that would run on the same looper thread. These two events, which can be from different software event queues, would execute in the ESP modes of that core. This scheme works for most events. In some infrequent cases, however, the events might not execute in the predicted order. For example, if a synchronous barrier is posted to an event queue, it would hold up processing of all subsequent synchronous tasks, while allowing later asynchronous tasks to execute ahead of them. In these cases, the information gathered for the predicted event, then must not be used. An "incorrect prediction" bit in each entry of the hardware event queue, would regulate whether the information in I, D and B-lists should be used.

5. Methodology

We evaluate ESP using seven real asynchronous web applications (Figure 6). Our benchmark suite has a representative application for each of the different types of web applications commonly used, ranging from e-commerce to social networking to online image editing. Each of our browsing sessions are short but complete. They capture typical user behaviors. We chose not to consider JavaScript (JS) benchmark suites such as Octane [1] and Sunspider [2], as they are shown to not resemble real world behavior of JS web applications [28].

We modified the V8 JavaScript engine in Chromium to identify JavaScript event executions within the browser's renderer process. In order to create workloads that are repeatable, we used the trace-recording component of SniperSim [8] to collect instruction traces for Chromium's renderer process running on Ubuntu 12.04. The trace captures JavaScript event execution, including the native code executed as part of library calls. These traces were used to simulate events executing in the normal mode.

Also, during our browsing sessions, before the start of every event, the next two events were executed in separate forked-off Chromium renderer processes. The instruction traces thus recorded, were used to simulate events executing in ESP modes.

Web Site	Actions performed	# events executed	# inst. (millions)
amazon (amazon.com)	Search for a pair of headphones, click on one result, go to a related item	7,787	434
bing (bing.com)	Search for the term “Roger Federer”, go to new results	4,858	259
cnn (cnn.com)	Click on the headline, go to world news	13,409	1,230
fb (facebook.com)	Visit own homepage, go to communities, go to pictures	9,305	2,165
gmaps (maps.google. com)	Search for two addresses, get driving, public transit directions, biking directions	7,298	2,722
gdocs (docs.google.com)	Open a spreadsheet, insert data, add 5 values	1,714	809
pixlr (pixlr.com)	Add various filters to an image uploaded from the computer	465	26

Figure 6: Benchmark Web Applications.

Core	4-wide, 1.66 GHz OoO, 96-entry ROB, 16-entry LSQ
L1-(I,D)-Cache	32 KB, 2-way, 64 B lines, 2 cycle hit latency, LRU
L2 Cache	2 MB, 16-way, 64 B lines, 21 cycle hit latency, LRU
Main Memory	4 GB DRAM, 101 cycle access latency, 12.8 GB/s bandwidth
Branch Predictor	Pentium M branch predictor 15 cycle mispredict penalty 2k-entry Global Predictor, 256-entry iBTB, 2k-entry BTB, 256-entry Loop Branch Predictor, 4k-entry Local Predictor
Prefetchers	Instruction: Next-line (NL); Data: NL, Stride (256 entries)
Interconnect	Bus
Energy modeling	V_{dd} = 1.2 V, 32 nm

Figure 7: Simulator Configuration.

More than 98% of all events in forked-off Chromium processes executed to completion. These events matched their non-speculative counterparts with a greater than 99% accuracy, validating our claim that events in these applications are largely independent of each other. The remaining events failed when they veered off the correct non-speculative path.

We modified SniperSim to model our ESP architecture, Runahead execution [27], and next-line (NL) prefetchers for instruction [5] and data caches. Our next-line prefetcher for the data cache is modelled similar to Intel's DCU prefetcher [15], which waits for four consecutive accesses to the same data cache line before prefetching the next. Our baseline processor (Figure 7) is modelled similar to the one used in Samsung's Exynos 5250 ARM-based System-on-Chip (SoC). ESP architecture configuration is shown in Figure 8. ESP adds 13.8 KB of hardware state to baseline.

HW structure	Description	ESP-1	ESP-2
L1-(I,D) Cachelet	12-way, 64 B lines, 2 cycle hit latency, LRU	5.5 KB	0.5 KB
I-List	Circular Queue	499 B	68 B
D-List	Circular Queue	510 B	57 B
B-List-Direction	Circular Queue	566 B	80 B
B-List-Target	Circular Queue	41 B	6 B
RRAT	32-entry RAT	28 B	28 B
HW Event Queue	2-entry queue	8 B	8 B
Special Registers	PC, SP, Flags, ESP-mode	12 B	12 B
All HW additions		**12.6 KB**	**1.2 KB**

Figure 8: ESP Hardware Configuration.

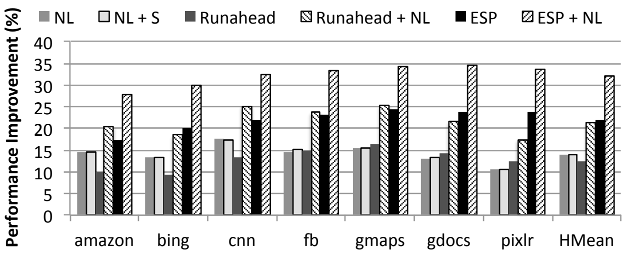


Figure 9: Performance of ESP, Next-Line and Runahead.

We evaluated the energy expended by the different designs using McPAT 1.2 [22]. CACTI 5.3 [33] was used for the added cache-like structures.

6. Results

We evaluate our ESP architecture and compare it to the runahead execution [27]. We also show that it complements next-line (NL) prefetchers. Later, we analyze ESP's effectiveness in improving I-cache, D-cache and branch predictor performance separately. Finally, we analyze the energy overhead due to additional instructions executed in ESP.

Our ESP architecture improves I-cache (I), D-cache (D), and branch prediction (B) performance. We study the benefits due to each of these optimizations separately. The suffixes (I, D, B) that we use to ESP indicate the presence of one of the three optimizations.

6.1. ESP Vs Next-Line Vs Runahead

Figure 9 compares performance of ESP with next-line (NL) prefetching, data stride prefetching (S) and runahead execution. Results are normalized to the baseline with no prefetching. It also shows the benefits of combining NL with ESP or Runahead execution. For these workloads, data stride prefetching provides an almost insignificant performance benefit of 0.1% over NL.

We find that our ESP design with next-line prefetching can improve performance by about 32% relative to the baseline, and 16% compared to NL + S. In contrast, Runahead with next-line provides only 21% performance improvement over baseline.

Next-line prefetching is a simple solution, and still provides nearly 13.8% performance benefit over no prefetching. However, its effectiveness is limited as it relies on spatial locality, which is not significant in most asynchronous applications. An interesting result is that NL can complement our ESP optimization very well. For long running events, ESP may not have sufficient opportunity to pre-execute deep into those events. For such events, when ESP prediction is not available, simple NL performs useful prefetching. We expect that other

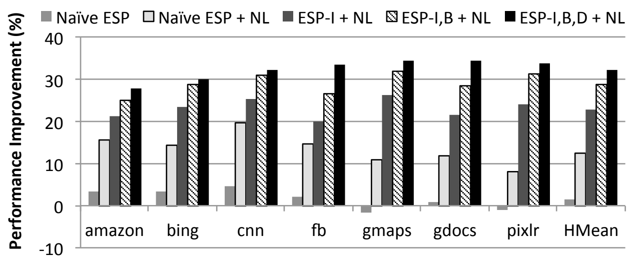


Figure 10: Sources of Performance in ESP.

prefetchers could similarly complement ESP.

Runahead by itself provides only 12% improvement over baseline. As we discussed earlier in Section 1, Runahead does not help improve performance in the presence of I-cache misses. As we discussed in Figure 3, reducing penalty due to I-cache misses is significantly more important than D-cache misses. Also, Runahead's effectiveness relies on discovering independent instructions after a load miss, which is a significant limitation. In contrast, ESP can jump ahead to the next event and easily discover independent computation to pre-execute. Nevertheless, we find that NL instruction prefetcher can complement Runahead as well as it complements ESP.

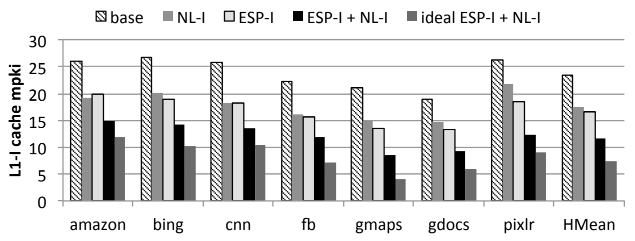
6.2. Sources of Performance in ESP

Figure 10 shows the sources of performance in ESP. To study the need for cachelets and predictor lists, we consider a hypothetical naive ESP design that does not use these extra structures. Instead, it fetches cache blocks into regular memory hierarchy (L1, L2) and updates branch predictors, during event pre-executions. This design is similar in principle to Runahead designs [16, 7, 26, 25], which instantly prefetches cache blocks and updates branch predictors in the runahead mode. We find that this naive ESP design hardly improves performance. In fact, it degrades performance for some applications (`pixlr`). This is because ESP jumps much father ahead into the execution than runahead. Therefore, prefetching blocks instantly during pre-executions would be too early. Prefetched blocks may get evicted before use, and even worse, it can degrade performance by evicting useful cache blocks used by the current non-speculative event.

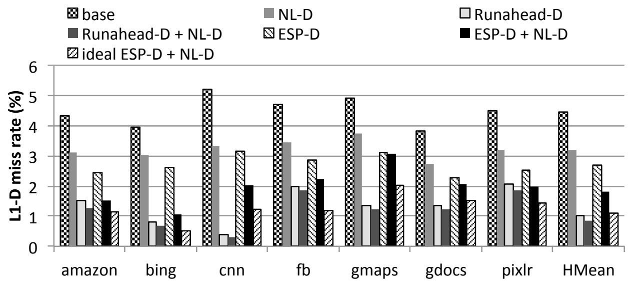
Figure 10 also shows that I-cache prefetches initiated through information recorded in I-lists provides the most significant benefit (9.1%), over and above NL. Branch prediction also adds appreciable performance gain (6%), but D-cache prefetching provides only marginal benefit (3.3%). This is not surprising given that the highest performance potential was by reducing I-cache misses (Figure 3). In the following sub-sections, we analyze each of these benefits in more depth.

6.3. ESP for Instruction Cache Performance

Figure 11a shows L1 I-cache misses per kilo-instructions (MPKI) for various configurations. NL-I represents next-line prefetching for I-cache only. ESP along with next-line reduces I-cache MPKI from about 23.5 to 11.6. To understand the



(a) I-cache performance.



(b) D-cache performance

Figure 11: Cache Performance

potential of ideal ESP design, we evaluated an ESP configuration with infinite I-cachelet and I-list, and also assumed timely prefetches into the L1 I-cache. Our design comes close to ideal.

6.4. ESP for Data Cache Performance

Figure 11b shows L1 D-cache miss rate for Runahead-D, ESP-D, and their combinations with next-line data-only prefetcher (NL-D). Runahead-D is Runahead Execution but only warms up the data cache, but does not update branch predictors.

Our ESP architecture with NL-D reduces L1 D-cache miss rate from 4.4% to 1.8%, whereas Runahead with NL-D does slightly better by reducing it to 0.8%. As expected, Runahead execution performs well for D-cache performance, since it is able to warm-up the data cache in short bursts (during LLC data misses) throughout the execution of the event. ESP-D design is relatively less effective. However, an ideal ESP-D design, with infinite cachelet size and timely prefetches, performs comparably to Runahead. This demonstrates that there is room for improving ESP design further for data cache performance.

6.5. ESP for Branch Predictor Performance

ESP reduces branch misprediction rate from 9.9% to 6.1% (Figure 12). One important design question is how to update branch predictor states during pre-execution (ESP mode). Not updating the branch predictor would compromise pre-execution's ILP, whereas updating them may compromise normal event's ILP. The other problem is how to learn from the mispredictions during a pre-execution of an event so that they are prevented during its normal execution.

We carefully navigated the design space to arrive at the design discussed in Section 4.3. Figure 12 shows the branch

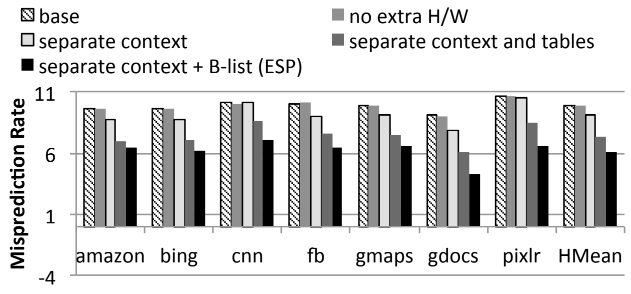


Figure 12: Branch misprediction rate.

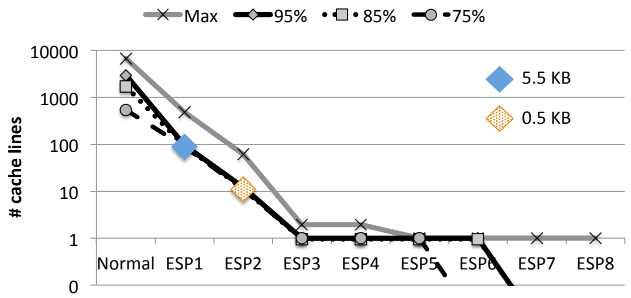


Figure 13: I-cachelet Size.

misprediction rate for different configurations that we considered. Naively updating branch predictor state during pre-executions did not provide any gains. At another extreme, we considered replicating all the branch prediction hardware for each ESP mode ("separate context and tables"). By warming up the replicated branch predictor in the ESP mode, and using its state during the event's normal execution reduced branch misprediction rate from 9.9% to 7.4%. However, this design is clearly area inefficient.

In the ESP design, we just replicated the branch prediction "context", which is the Path Information Register (PIR) used to index into the predictor tables. Predictor tables were updated in the ESP modes. This helped us avoid significant interference between normal and the ESP modes. Coupled with the B-list that enabled us to train the branch predictor with a preset number of branches ahead of the current branch during the normal execution, our design even managed to outperform the design that replicated all branch predictor hardware (6.1% as opposed to 7.4%).

6.6. Area Overhead

We define working set of a pre-executed event to be the number of cache blocks that need to cached to exploit all reuse. Sizes of a instruction (I) or a data (D) cachelet need to be large enough to keep track of the working set of pre-executed events.

Figure 13 shows the max working set of all events executed in each of the different ESP modes in logarithmic scale. For comparison, it also shows the max working set of all the events executed in the normal mode ("Normal"). As pre-execution of events are relatively shorter than full normal execution, their working set sizes are order of magnitude smaller (e.g., Max of "ESP1" is 10x smaller than max of "Normal").

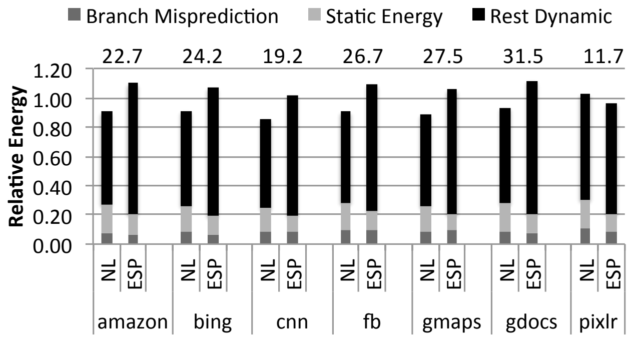


Figure 14: Energy Overhead.

It is however wasteful to provision cachelets to support worst case requirements. Therefore, we estimate cachelet size required to capture 95% of reuse in 95% of all events. We find that we could further reduce cachelet size requirement by another order of magnitude to about 5.5 KB for ESP-1. Cachelet size required for higher ESP-2 mode (jump ahead two events) is even smaller (0.5 KB), as events spend much less time in this mode.

These results also show that it is wasteful to provision higher order ESP modes beyond ESP-2, because we rarely see any opportunity to touch a cache block in these modes. We omit results for D-cachelet size, but the conclusions were similar to I-cachelets.

Figure 8 lists most additional hardware structures used in ESP to support two modes. ESP-1 mode requires 12.6 KB, whereas ESP-2 mode only requires 1.2 KB. In addition to these, ESP has logic, control and wiring overheads, though the hardware overhead of these is extremely small compared to the aggregate mentioned in Figure 8.

6.7. Energy Overhead

ESP uses additional hardware (Figure 8), and also executes on average 21.2% more instructions than the baseline, which can potentially increase the energy overhead. However, since it reduces latency, it can reduce static energy consumption. Branch mispredictions are also reduced, leading to less dynamic energy spent in executing wrong path instructions. Combining the above factors, ESP ends up consuming about 8% more energy than the baseline as shown in Figure 14. Numbers on top of the bars specify the percentage of additional instructions executed compared to the baseline.

7. Related Work

Unlike ESP, past work had not attempted to optimize general-purpose processor architectures for asynchronous programs in general. Runahead execution [16, 7, 26, 25] is perhaps the most closely related work. Sections 1 and 6 provided an in-depth comparison to this optimization.

Speculative Pre-Execution: Helper threads execute just the backward slices of the program required to compute data addresses and branch outcomes to fetch data cache blocks early and warm-up the branch predictor [38, 29, 14, 10, 32]. Compared to ESP, these techniques share similar limitations that we discussed for runahead, such as not being able to resolve I-cache misses. Also, unlike these techniques and runahead execution, ESP speculatively pre-executes instructions from future events, thus making it more effective in finding instructions independent from the current event. However, since the next event may start execution thousands to millions of instructions later, simply warming up the caches and the branch predictor does not help as it is too premature. This requires new solutions, as described earlier.

CGMT/SMT/Idle Cores: Additional execution contexts in coarse-grained switch-on-event multithreading (CGMT) [4] or SMT [34] could potentially be used to speculatively pre-execute future events. However, they may also need ESP-like versioning support to isolate cache and branch predictor state of current non-speculative event from pre-execution of future events.

Using a thread context on an idle core, however, will not help us utilize idle cycles in a core due to long latency cache misses. It would also incur additional overhead to transfer live-ins to the core that pre-executes future event from the non-speculative event's core, and then transfer back the information gathered during pre-execution.

Optimistic Concurrency: Going beyond speculative pre-execution and ESP, events could potentially be executed concurrently, optimistically assuming they are independent, which is verified at the end of event execution. If so, the work done is committed. In addition to extra execution context and versioning support, optimistic concurrency (e.g. TLS [19, 23, 30, 31], TM [21]) would require significant hardware support to check for dependencies between concurrently executed events and recover from mis-speculations. Furthermore, we discovered that, though events are logically independent of each other, there were a small number of memory dependencies between most events. Therefore, naïvely checking for memory dependency conflicts (using transactional memory like hardware) would be too conservative and may lead to serialization of most events.

Thus, ESP makes a strong case for exploiting event-level parallelism using speculative pre-execution, however, it may be possible to exploit this parallelism more aggressively in the future.

Prefetching: In Section 6, we showed that next-line prefetching [5] can complement ESP. Other instruction prefetchers can complement ESP well too. However, as explained in Sections 2.1 and 2.3, conventional instruction prefetchers perform poorly.

Compared to a recent instruction prefetcher, EFetch [9], ESP incurs 3x less hardware overhead and attains 6% higher performance over the common baseline of no prefetching. Compared to PIF [18], ESP incurs 15x less hardware overhead and attains 10% higher performance.

ESP also helps improve branch predictor and data cache performance. While there has been significant work on data prefetching [6, 11, 12], potential for performance gains from data cache optimization is limited (Figure 3).

Web Browser Optimizations: Crom [24] is a software solution that exploits event-level parallelism by speculatively executing events when the browser is idle using cloned browser context. ESP is a processor solution for hiding microarchitectural bottlenecks in asynchronous programs.

Zhu and Reddi [36] exploit slack in loading web page to save energy by executing on energy-efficient "little" cores. They also developed custom accelerators for style-resolution kernel and a specialized cache for tracking important data structures in the web browser [37]. In contrast, ESP is a more general-purpose architecture that aims to improve the performance of all asynchronous applications.

Domain-Specific Accelerators: Like ESP, some previous works have identified the opportunities provided by event-driven sensor network applications and have designed energy-efficient specialized accelerators [20, 17] for these applications. However, unlike ESP, these techniques are very specific to one domain of asynchronous programs (sensor networks) and improve energy-efficiency. ESP can be used to improve performance of a wide range of asynchronous programs including sensor networks.

8. Conclusion

Conventional processor architectures have largely ignored the characteristics of asynchronous programs, which constitute a large fraction of all computing systems built today. As the processor industry embraces heterogeneity, we propose that some cores in a multi-core processor be enhanced to improve the efficiency of asynchronous program execution.

Towards this end, this paper presented the ESP architecture. ESP peeks into the event queue to determine pending future events. It exploits this knowledge to profitably pre-execute future events in the presence of long latency cache misses. Pre-execution of an event helps ESP determine instruction and data addresses that need to be prefetched and branch mispredictions that need to be corrected. This information is later used to remove the microarchitectural roadblocks while executing an event in the normal mode.

We showed that ESP can improve the performance of seven real web applications by an average of 16%, whereas traditional runahead only achieves a 6.4% gain. We expect comparable performance improvements for other asynchronous systems such as the mobile applications with similar characteristics as the web applications.

9. Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments and feedback. This research was supported by the National Science Foundation grants SHF-1217917 and CAREER-1149773.

References

[1] "Octane benchmark suite," https://developers.google.com/octane/.

[2] "Sunspider benchmark suite," https://www.webkit.org/perf/sunspider/sunspider.html.

[3] "Why threads are a bad idea," http://web.stanford.edu/ ouster/cgi-bin/papers/threads.pdf.

[4] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin, "Sparcle: An evolutionary processor design for large-scale multiprocessors," *Micro, IEEE*, vol. 13, no. 3, pp. 48–61, 1993.

[5] D. Anderson, F. Sparacio, and R. M. Tomasulo, "The ibm system/360 model 91: Machine philosophy and instruction-handling," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 8–24, 1967.

[6] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*. IEEE, 2001, pp. 52–61.

[7] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Dynamically allocating processor resources between nearby and distant ilp," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*. IEEE, 2001, pp. 26–37.

[8] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov. 2011.

[9] G. Chadha, S. Mahlke, and S. Narayanasamy, "Efetch: optimizing instruction fetch for event-driven web applications," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 75–86.

[10] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (ssmt)," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 186–195. [Online]. Available: http://dx.doi.org/10.1145/300979.300995

[11] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Transactions on*, vol. 44, no. 5, pp. 609–623, 1995.

[12] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-m. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *Proceedings of the 24th annual international symposium on Microarchitecture*. ACM, 1991, pp. 69–73.

[13] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *Proceedings of the 11th symposium on Operating Systems Design and Implementation*, 2014.

[14] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: ACM, 2001, pp. 14–25. [Online]. Available: http://doi.acm.org/10.1145/379240.379248

[15] J. Doweck, "White paper inside intel® core$^{\text{TM}}$ microarchitecture and smart memory access," *Intel Corporation*, 2006.

[16] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th international conference on Supercomputing*. ACM, 1997, pp. 68–75.

[17] V. Ekanayake, C. Kelly IV, and R. Manohar, "An ultra low-power processor for sensor networks," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 5, pp. 27–36, 2004.

[18] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. New York, NY, USA: ACM, 2011, pp. 152–162. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155638

[19] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII. New York, NY, USA: ACM, 1998, pp. 58–69. [Online]. Available: http://doi.acm.org/10.1145/291069.291020

[20] M. Hempstead, N. Tripathi, P. Mauro, G.-Y. Wei, and D. Brooks, "An ultra low power system architecture for sensor network applications," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 208–219.

[21] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*. ACM, 1993, vol. 21, no. 2.

[22] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 2009, pp. 469–480.

[23] J. F. Martínez and J. Torrellas, "Speculative synchronization: applying thread-level speculation to explicitly parallel applications," in *ACM SIGOPS Operating Systems Review*, vol. 36, no. 5. ACM, 2002, pp. 18–29.

[24] J. Mickens, J. Elson, J. Howell, and J. Lorch, "Crom: Faster web browsing using speculative execution," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. USENIX Association, 2010, pp. 9–9.

[25] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for efficient processing in runahead execution engines," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 370–381. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2005.49

[26] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003, pp. 129–140.

[27] ——, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003, pp. 129–140.

[28] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "Jsmeter: Comparing the behavior of javascript benchmarks with real web applications," in *Proceedings of the 2010 USENIX conference on Web application development*. USENIX Association, 2010, pp. 3–3.

[29] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE, 2001, pp. 37–48.

[30] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 414–425. [Online]. Available: http://doi.acm.org/10.1145/223982.224451

[31] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/339647.339650

[32] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: improving both performance and fault tolerance," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 257–268, 2000.

[33] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi, "Cacti 5.3," *HP Laboratories, Palo Alto, CA*, 2008.

[34] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 392–403.

[35] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 207–217.

[36] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 13–24.

[37] ——, "Webcore: architectural support for mobileweb browsing," in *Proceeding of the 41st annual international symposium on Computer architecuture*. IEEE Press, 2014, pp. 541–552.

[38] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*. IEEE, 2001, pp. 2–13.